Locks and Barriers in Checkpointing and Recovery

Ramamurthy Badrinath, Christine Morin

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Locks and Barriers in Checkpointing and Recovery

Ramamurthy Badrinath and Christine Morin

N°5021

Novembre 2003

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--5021--FR+ENG

Locks and Barriers in Checkpointing and Recovery

Ramamurthy Badrinath * and Christine Morin †

Thème 1 — Réseaux et systèmes
Projet PARIS

Rapport de recherche n ° 5021 — Novembre 2003 — 14 pages

Abstract: Dependency tracking between communicating tasks is an important concept in backward error recovery for parallel applications. One can extend the traditional dependence tracking model for message passing systems to track dependencies between shared memory and task private states for shared memory applications. The objective of this paper is to analyze the issues generated by locks and barriers in parallel applications so that we can checkpoint tasks at any time (even when holding or waiting for locks and barriers). In particular we attempt to extend earlier dependency tracking mechanisms to locks and barriers. We address both coordinated and uncoordinated checkpointing schemes.

Key-words: Lock, barrier, synchronization, checkpointing, backward error recovery, dependency tracking, cluster

(Résumé : tsvp)

* badrinar@india.hp.com
† Christine.Morin@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Verrous et barrières dans les protocoles de sauvegarde et restauration de points de reprise

Résumé : La détection des dépendances entre des tâches communicantes est un concept important dans les protocoles de recouvrement arrière pour les applications parallèles. Il est possible d'étendre le modèle de détection de dépendances traditionnel dans les systèmes à échange de messages afin de détecter les dépendances entre les états privés des tâches et la mémoire partagée pour les applications parallèles fondées sur le paradigme de communication par mémoire partagée. L'objectif de cet article est d'analyser les problèmes soulevés par les verrous et les barrières dans les applications parallèles lorsqu'il s'agit de pouvoir sauvegarder un point de reprise à tout momemnt (même pendant qu'une tâche détient un verrou ou que des tâches sont bloquées sur une barrière). En particulier, nous tentons d'étendre les mécanismes de détection de dépendances existants pour prendre en compte les verrous et les barrières. Nous nous intéressons aux stratégies de sauvegarde coordonnée et non coordonnée de points de reprise de processus.

Mots-clé : Verrou, barrière, synchronisation, point de reprise, recouvrement arrière, détection de dépendances, grappe

1 Introduction

As cluster computing systems supporting the DSM programming model come to be more widespread in use, supporting fault tolerance in these systems has attracted a lot of interest. Several works have dealt with evaluating the overhead of various checkpointing protocols ([12] provides a survey); they have dealt with tracking dependencies[15, 3] and supporting some checkpoint optimizations[8, 10].

Several systems for checkpointing and recovery of applications running on DSM models have been described in literature. Yet, in practice, DSM systems over commodity hardware that support fault tolerance have been scarce. While [8, 10] deal implicitly with sequential consistency models, coordinated checkpointing and recovery by rollback, [14, 6, 13] and [5] use release consistency, uncoordinated checkpointing and recovery by replaying some log information. [11] describes coordinated checkpointing in the context of a system using LRC consistency model.

Several of these works such as [9] also recognize the issues of reconstructing the state of locks or barriers, and the state of the DSM machine. In [8] the authors realize the importance of providing negative acknowledgments or *aborts* when it is detected that a DSM operation or a synchronization primitive cannot be completed in the presence of a fault.

In [4] the authors use barriers in order to lower synchronization overheads in checkpointing in a sequentially consistent DSM with coordinated checkpointing. In the release consistency model, with uncoordinated checkpointing as in [14, 6, 13] it is natural to use locks in order to checkpoint tasks and memory states.

In this paper we study the impact of locks and barriers on checkpointing and computation of recovery lines in a sequential consistent system such as Kerrighed[2] which supports both DSM as well as message passing paradigms. Kerrighed is a system which uses kernel level support for checkpointing and recovery. In [2] we propose coordinated checkpointing for Kerrighed and evaluate its performance. The techniques presented there do not handle locks and barrier issues in the larger context of possible use in uncoordinated checkpointing, or optimized coordinated checkpointing and recovery. In this paper we try to study the dependencies generated among tasks because of locks and barriers. Our aim is to be able to implement mechanisms that will allow us to support, for instance, system initiated checkpointing with the lock or barrier in an arbitrary state. We show how this impacts uncoordinated and coordinated checkpointing and recovery.

In Section 2 we describe the general dependency tracking mechanism in brief. In Section 3 and Section 4 we describe the issues concerning locks and barriers respectively showing in each case how they may be addressed. Finally we conclude in Section 5 with some observations.

2 Dependency Tracking

We consider a parallel application with multiple tasks which communicate by explicit message exchange (through *send* and *receive* primitives), or by accessing shared memory (through

read and *write* operations to words in shared memory). In addition tasks also interact through barrier synchronisation and locks (*acquire* and *release* operations). For this paper we consider the dependencies generated through shared memory at the page granularity level. Our reason for this choice is that page granularty is supported well by the underlying architecture. This choice does not change the issues of dependency tracking discussed herein.

Inter process communication (IPC) in any form causes dependencies, among the tasks that are communicating, that must be honored when computing a globally consistent state for recovery. Hence in a system to provide fault tolerance and to support optimizations of fault tolerance protocols it is important to provide mechanisms to track dependencies. The dependencies that are generated are analyzed during the checkpoint or recovery phase and are an essential ingredient to computing recovery lines. For instance suppose p_1 writes a value to a variable which p_2 reads. If due to faults p_1 is rolled back to a state prior to the write then p_2 needs to be rolled back to a state prior to the read. p_1 and p_2 are respectively referred to as the source and destination of the dependency generated.

In practice these mechanisms need to be particularly light-weight since they may be activated whenever there is IPC adding to the overhead of IPC.

The actual use of the mechanisms described below depends on the checkpointing and recovery scheme implemented. In [7] one finds a description of various checkpointing and recovery strategies. [1] provides a detailed account of the dependencies generated by message passing and shared memory access, and discusses how the mechanisms may be used to support a variety of checkpointing and recovery protocols. In this section we intend to summarize a description of those mechanisms.

We propose to track direct dependencies between memory elements (pages) and tasks, and among tasks. Thus pages and tasks are the basic entities among which we track dependencies. We describe an outline of the mechanism below and refer the reader to [1] for details, with examples, of their usage with respect to reads and writes to shared memory. This is an extention of the mechanisms proposed in [3] to shared memory pages. We assume each entity in the system (i.e., each page and each task) has a distinct identity. We associate with each entity the following attributes:

1. An integer called a *sequence number* (**sn**). This is initialized to 1 when the entity begins its existence. This is incremented only when the entity takes a checkpoint. Thus a sequence number can be used to distinguish intervals between checkpoints. If in an interval is the value of the sequence number is x, we will refer to it as interval number x. The sequence number of the source of a dependency is delivered to the destination of the dependency whenever a dependency is created. The local value of sn will always be the *checkpoint number* of the next checkpoint to be taken for the entity. We will denote by $c_{i,j}$ the j^{th} checkpoint taken by entity i.

2. A vector called the *direct dependency vector* (**DDV**). Over time, due to IPC, an entity receives a number of sequence numbers from other entities. These are stored in the DDV of the entity, thus recording dependencies. Initially for an entity j, $ddv[j]$

contains all zeros. If there is a dependency from entity i to j (for instance a message is sent from task i is received by task j), and the sequence number (of i) sent in the interaction is x, and $ddv[j]$ is the local dependency vector with entity j, then on the event at j, we execute the following code:

$$ddv[j][i] = \max\{ddv[j][i], x\}$$

Whenever we checkpoint an entity j we store along with the local checkpoint, the corresponding DDV, i.e., $ddv[j]$. This saved DDV is called the *timestamp* of the corresponding checkpoint. Whenever a checkpoint is taken at a node j we execute the following code:

$$ddv[j][j] = sn; sn++;$$
$$\texttt{save the timestamp } ddv[j].$$

We will refer to the timestamp associated with the k^{th} checkpoint of entity j as $ddv_k[j]$. Note that $ddv_k[j][j] == k$. For completeness sake, the timestamp for the zero-th checkpoint for all entities has all zeros.

This mechanism essentially records direct dependencies between checkpoints of various entities. So for instance if entity i decides to rollback to checkpoint number n, then for entity j if $ddv[j][i] > n$, then clearly entity j needs to rollback to a checkpointed state for which $ddv[j][i] \leq n$. Of course this covers only direct dependencies. The recovery line computation is the responsibility of the recovery protocol. In [3, 15] it is shown that tracking direct dependencies suffices for computing the recovery line.

With this mechanism in place it is possible to detect all direct dependencies. Yet, in the case of shared memory, some optimization is possible. Consider a page that has not been changed between the two checkpoints $c_{i,j}$ and $c_{i,k}(k > j)$ for the page, then clearly any read after $c_{i,k}$ still refers to the version checkpointed at $c_{i,j}$ provided the page has not yet been modified before the read after $c_{i,k}$. Recording the newer sequence number by the reader results in an artificial dependency. Hence in a read of a page we may prefer to use an older value than the actual sequence number of the page. For this we introduce the following third attribute for each page:

3. An integer counter called the *last write number* (**lwn**). This maintains the interval number of the last interval during which the page was written to. On a read the *lwn* of the page is 'received' by the reading task rather than the *sn* of the page. Also this means that on write to a page, the page must update *lwn* to its current value of *sn*.

One uses the dependency information captured by these attributes during the analysis of the fault just prior to recovery. One can also use the *lwn* and *sn* to support incremental checkpointing. The value of the expression $sn == lwn$ evaluates to `true` at a checkpoint if and only if the page has been modified since the last checkpoint.

In the next two sections we see if and how locks and barriers may be integrated as entities in this scheme.

3 Lock States and Dependencies

Parallel applications commonly use locks in order to implement mutual exclusion. We consider the two common operations on a lock, namely, *acquire* and *release*. A task that successfully returns from an *acquire* operation is said to be *holding* the lock until it performs a *release* on the same lock. The primitives ensure that at any time at most one task is holding the lock.

A lock state can be described by the following two variables:

1. `holder` This variable indicates the identifier (also called PID) of the task that has currently successfully acquired the lock and not yet released it.

2. `waiters` This is a list of PIDs of tasks that have executed a request to acquire, but have not yet succeeded, and hence are waiting to acquire.

If R represents a recovery line, l a lock, and $T(R, l)$ is the subset of tasks of R that hold a lock l, the constraint imposed by locks on R can be stated as:

Lock Rule: $\forall l : T(R, l)$ has size at most 1.

If $T(R, l)$ has more than one task, we say that these tasks conflict on the lock l.

This implies that as far as locks are concerned, there is a problem if and only if in the recovery line two or more tasks conflict in the ownership of some lock l. This is only possible if rollback states can be states where locks are being held. Notice that there is no problem if a rollback state is one where a task is in a lock's wait queue. Of course we need mechanisms to checkpoint and restart tasks in a lock wait queue and to re-initialize lock ownership state.

We now ask if we can treat the lock as a checkpointable entity and introduce dependencies between locks and tasks so as to achieve a recovery line that takes lock conflicts into account. We note that the only interactions involving a lock are the acquire and release operations. The only lock states that are of interest to us are states where a lock is being held by a task. We denote by l_i the state of the lock l after the i^{th} acquire.

One can treat every successful acquire and release as a two way dependency between a lock and a task. For instance, one may model the lock as a shared memory containing the attributes mentioned earlier, and treat every access and release as memory writes, since they change the attribute values. This creates sufficient dependencies but will create a lot of false (or unwanted) dependencies. Additionally it throws up the question of the checkpointable state where a request for a lock has been made but the acquire itself has not yet succeeded. To understand false dependencies consider this example - A task acquires and releases a lock and *all checkpoints* are *before the acquire* and *after the release*, there is no real dependence between the lock and the task, either can rollback without effecting the other, whereas the DDV will record that there are dependencies both ways. It is important to note that such false dependencies are essentially caused if we assume that successive instances of holding the lock, l_i and l_{i+1} are causally related. Using the traditional notation (eg., from [1]) we express this assumption as $l_i \rightarrow l_{i+1}$. Further since only one task is involved in a lock at

a time, the *lock itself causes no inter-process dependency.* Thus we avoid associating DDVs with locks, and treating them as separate checkpointable entities.

Traditionally the dependencies between task states is represented as a graph with checkpointed states being the vertices and edges representing task states. Such a graph is called a *checkpoint graph*[7]. In [1] this graph is extended to included checkpointed shared page states. What we need is to maintain information on locks held at checkpointed states. Traditional dependence analysis (eg., as in [7]) requires a traversal of the checkpoint graph to determine a valid recovery line. We need to verify the lock constraint as supplement to this traditional traversal computation on the checkpoint graph. All lock states are reconstructed and a lock is not treated as a separate entity which can independently rollback. Thus *the only inconsistency introduced by locks in the recovery line is an issue of resource contention.*

In an **uncoordinated** case, rollback could lead to a domino-like effect (but not really) even considering only lock dependencies (no other IPC, which seems unlikely). This can be shown in a case with two tasks that acquire locks, and where all checkpoints of both tasks are such that they posses the lock at the checkpoint, the failure occurring in the task that is currently not holding the lock. Here one task will have to be rolled back to the initial state because only then can we have a recovery line with no contention on the lock! This one task can be either. In fact, a latest recovery line is not uniquely defined.

One way to do uncoordinated checkpointing and still avoid problems of contention in rollback is if one detects that a task to be checkpointed is holding a lock and delays checkpointing until the release that leads to a state where no locks are held. In this case one is sure never to rollback to a state where a lock is held. Thus traditional rollback algorithms suffice. The only problem with this scheme is that a certain checkpoint interval cannot be guaranteed. We will now assume that we wish indeed to be able to checkpoint the task at any time.

We present in Algorithm 1 a heuristic to compute a recovery line. There are essentially two steps marked S_1 and S_2 which resolve dependency related and resource contention respectively. The function ComputeRecoveryLine is the traditional checkpoint graph traversal recovery line computation algorithm; it operates on C, the traditional checkpoint graph, and R, the latest set of states available for each task. Then we compute L_c, the set of locks which are in contention in the newly computed recovery line R. The termination condition is that that this set $L_c(R)$ be empty, i.e., we have a recovery line with no causality conflicts as well as no lock contention. The function ResolveLockContention resolves contention for locks of $L_c(R)$. The result is a modified potential recovery line R which may violate causality. Hence we need to re-execute the loop.

Since there is no unique latest recovery line, several heuristics may be used by ResolveLockContention to select the set of tasks in R to rollback. As examples, one may rollback all but one tasks from the set $T(R, l)$ arbitrarily, one may rollback only one task, one may rollback the task which has the fewest number of previous checkpoints, etc. Also, one may choose to execute the step S_2 for all locks in L_c or only for one lock, before proceeding to the next iteration on ComputeRecoveryLine.

One may also note that the dependency due to locks may require similar modifications to any garbage collection algorithms that may be used in the case of uncoordinated checkpointing.

```
   input    : A checkpoint graph C.
   output   : A valid recovery line R.
   begin
      R = Latest set of recoverable points of all tasks;
      while true do
S1       R =ComputeRecoveryLine (R, C);
         Compute L_c(R) to be the set of locks with contention in R;
         if L_c(R) is empty then return R ;
S2       R = ResolveLockContention (R, L_c) ;
      end
   end
```

Algorithm 1: Recovery from Uncoordinated Checkpoints with Locks

In the case of **coordinated** checkpointing during recovery we know that the lock can be reinitialized to the checkpointed state, provided of course *all* tasks will rollback/restart. We may want to optimize and decide not to rollback all tasks, only those which have dependency on the post checkpoint activity of the failed tasks. For locks this means that if t_i rolls back to a state where it holds lock L, and another task t_j holds the lock L at failure detection time, then t_j must be rolled back to its latest checkpoint too. This checkpoint to which t_j rolls back will be consistent with the rolled back state of t_i thanks to coordinated checkpointing. Thus at most one task will be forced to rollback for maintaining lock consistency (per lock).

To conclude this section, locks do not require us to introduce a new entity into our model. They do not cause inter task causal dependencies. They do however require us to do resource conflict resolution on recovery. In order to implement checkpointing for applications using locks we need mechanisms to detect locks being held by and awaited by a given task, and save this information along with the task's checkpoint. This information needs to be available to the recovery line computation module.

4 Barrier States and Dependencies

Barriers are a form of strong synchronization among tasks. At a barrier a certain number of tasks synchronise, and any task departs only after that certain number of tasks have arrived. We try to investigate the constraints posed by barriers on the recovery line. As before we may wish to extend our dependency tracking model to barriers. A barrier has the following state variables:

1. `degree`. The number of tasks to be awaited at the barrier (assumed static).

2. `arrived`. The list of PIDs of tasks that have arrived. If this is non empty it means that the barrier is active.

We note that it is not predefined as to which tasks will participate and whether the same set of tasks will always participate in the barrier. But in normal programming practice it is natural to have the same set of tasks arrive repeatedly at the barrier. Some times this is defined by some king of *group ID* for tasks arriving at the barrier. For the present analysis we do not make this assumption. Later we specialize to the case when the task set is known. In the general case we notice that just as for locks, successive invocations of the barrier are not by themselves dependent on each other. We explain this further after introducing the notation below for ease of presentation:

1. b_i: This is the i^{th} instance(invocation) of the usage of the barrier b.
2. $T(b_i)$: The set of tasks involved in the barrier b_i.
3. R denotes a potential recovery line and $R(T)$ is the set of states to which the tasks in some set T of tasks recovers (i.e., R restricted to the set T).
4. $f(b_i)$: *Future* of b_i: This is the set of all states of the tasks of $T(b_i)$ after arriving at the barrier b_i. Note that it includes task states which have left the barrier.
5. $p(b_i)$: *Past* of b_i: This is the set of all states of the tasks of $T(b_i)$ before departing from the barrier b_i. Note that this includes task states where some of the tasks have not yet arrived at the barrier. Thus there is a period where the future and the past of an invocation intersect. This is the duration when the i^{th} invocation is active. $sf(b_i) \stackrel{\Delta}{=} [f(b_i) - p(b_i)]$ and $sp(b_i) \stackrel{\Delta}{=} [p(b_i) - f(b_i)]$ respectively denote the *strict* future and *strict* past of b_i. We say $active(R, b_i)$ is true if and only if there is a task state in a potential recovery line R where some task is waiting on the barrier instance b_i.

The constraints imposed by barriers on a recovery line R can be stated as follows:

Barrier Rule 1: $\forall i, b : [R(T(b_i)) \subset f(b_i)] \bigvee [R(T(b_i)) \subset p(b_i)]$
Barrier Rule 2: $\forall b : \exists$ at most one i such that $active(R, b_i)$ holds.

The first rule says that for each instance of each barrier all recovery states must be on the same side (chronologically) of the barrier. Thus *a barrier instance b_i introduces dependencies between states of tasks in $T(b_i)$*. If for some recovery line R and some barrier instance b_i, the above property does not hold we say that the barrier and the recovery line conflict with each other.

Consider two instances of the invocation of the barrier b: with $T(b_i) = \{t_1, t_2\}$ and $T(b_{i+1}) = \{t_3, t_4\}$. If t_1 rolls back to a state in $sp(T(b_i))$, this does not imply rollback required for $T(b_{i+1})$. Thus, *successive invocations of the barrier are not by themselves dependent on each other*. We must make sure that if we use any graph traversal scheme then we do not incorporate $b_i \rightarrow b_{i+1}$ edges in the dependence analysis.

The second rule states that in any recovery line one would recover a barrier to at most one of its active invocation states. That is, if t_1 and t_2 in R are rolled back to states where they are in the wait queue of different instances of the same barrier b, then there is a conflict. Yet this is not dependence related. As for locks, *this is a resource conflict.*

One may supplement the dependence information in the DDVs as follows:

- For a task t: in addition to DDV, maintain the set $B = \{b_i : t \text{ left } b_i\}$ in the current checkpoint interval. One can view B as an extension of the DDV to barrier instances.
- For a barrier DDV: All tasks sequence numbers of tasks that entered the barrier. *We will checkpoint this after each successful invocation completes.* Note that this DDV is not cumulative over previous invocations of the barrier. Since we need to distinguish different instances of the barrier, we attach with each instance a barrier invocation number. This can be seen as analogous to the checkpoint sequence number introduced in the general model.

We note here that we need to expand our notion of traversal of the checkpoint graph augmented with barrier instance dependencies. When a task t rolls back the traversal will tell us all barrier instances to rollback, which will tell us all task states to rollback. Implementation of the recovery graph traversal may be speeded up using:

1. The DDV of a barrier to hint which sequence number the task may have been in when it exited the barrier. This is because, in the terminology of message passing based dependence analysis, the set of "senders" is equal to the set of "receivers" for a barrier, i.e., the set of tasks that arrived at a barrier instance is equal to the set of tasks that departed from that instance.
2. By maintaining an additional "cumulative DDV" with each barrier instance, so that it helps us avoid traversing pasts where there is no interaction with a certain instance of a barrier. This will state for each barrier instance the latest sequence numbers of tasks that arrived at the barrier. This is different from the DDV for a barrier instance mentioned above, because in this case the sequence number x for task t states that "the last time t arrived at the barrier, its sequence number was x".

During checkpointing we assume that we can checkpoint tasks which are waiting for the barrier to succeed. So we require mechanisms that allow checkpointing and re-initialization of a task in a barrier wait queue.

For recovery in the case of **uncoordinated checkpointing**, we have to deal with both the dependence as well as resource related conflicts. Algorithm 2 resolves this in two phases, just as the previous algorithm does for locks.

Again we use the traditional graph traversal algorithm, ComputeRecoveryLine for the dependency analysis, but we note that the input checkpoint graph is augmented in that it records dependencies due to barriers as well. We have discussed this above.

For the resource conflict resolution phase we need to use a heuristic as before. We compute $B_c(R)$ the set of barriers with more than one active instances in R. This can be

done by simply finding out for each task state in R what barrier instance it is waiting in (this is part of the checkpointed task state). If this set of barrier instances contains two separate instances b_i and b_j of the same barrier b, then the barrier b is put in the set B_c. The function ResolveBarrierContention will invalidate one more more instances of each barrier $b \in B_c$, thus producing a modified potential recovery line R. A heuristic that seems reasonable is to retain only the oldest invocation of the barrier (one with the minimum invocation number). But, as in the case for locks this may be any heuristic and we could resolve one or more barrier conflicts in R before trying to re-analyzing the dependency by checkpoint graph traversal.

We note that the issue addressed by this algorithm must also be handled by any garbage collection algorithm that may be used to prune the set of checkpoints to maintain.

```
   input   : A Checkpoint Graph C, augmented with barrier dependencies.
   output  : A Valid Recovery Line R.

   begin
      R = Latest set of recoverable points of all tasks;
      while true do
S1       R =ComputeRecoveryLine (R, C);
         Compute Bc(R), to be the set of barriers with more than one active instances in
         R ;
         if Bc(R) is empty then return (R) ;
S2       R =ResolveBarrierContention (R, Bc);
      end
   end
```

Algorithm 2: Recovery from Uncoordinated Checkpoints with Barriers.

Coordinated Checkpointing. In this case we know that reverting *all* tasks to the latest checkpoint will definitely work correctly. A possible optimization is to analyze and initiate only necessary rollbacks. For barriers this requires traversing the dependency graph to track and rollback all tasks with dependency through barriers (this is the traditional optimization method). It also requires resolving resource conflicts. This can be done by invoking the resource conflict resolution step (step S_2) of Algorithm 2. If in this step we merely rollback all tasks in the later invocation, we would have resolved resource conflicts because clearly earlier checkpoint states are consistent.

4.1 Specializations of Barriers

Although we do not find a common definition for barriers in literature, a common usage of barriers is to assume that the same set of tasks always come to a barrier. In that case the augmented checkpoint graph needs to be enhanced to include dependences between successive barrier instances (i.e., $b_i \rightarrow b_{i+1}$). In this case we do not need the resource conflict resolution step at all. The traditional recovery line computation on this graph produces the required recovery line.

4.2 Failed Barrier States

Since barrier states are small, one can maintain an available copy of each successfully completed instance of a barrier. These are nothing but barrier checkpoints, with the additional requirement that the barrier is checkpointed at each successful completion. Hence we do not have to deal with the possibility that we do not have a certain barrier invocation which may be needed in step S_2 in our algorithm.

Consider a barrier that is active at failure. Clearly this can cause only resource contention related rollback. This is handled by first reconstructing the failed barrier instance state from the live members waiting on the barrier, before proceeding with the resource contention analysis. If *all* tasks waiting on the barrier instance failed, then there is no problem, it is as if the barrier instance never existed.

We note however, that it is possible to reconstruct a previously completed state, even if the barrier checkpoint was lost. To do this, on recovery, each task declares the latest instance for each barrier that it is aware of (this information is in the DDV/ Checkpoint graph). We can look at the latest barrier instance checkpointed for the barrier. From this we can recompute all barrier instances in the future of the checkpointed instance. Assuming that the degree of the barrier is constant, it is possible to detect if each such reconstructed instance has a complete task set. If not, then that instance needs to be invalidated, causing corresponding traversal and rollback.

In conslusion, a barrier may be considered as a separate checkpointable entity. Barriers introduce causal dependencies between task states and checkpointed barrier instance states. Barriers also introduce resource conflicts which need to be addressed for recovery line computation. In order to support checkpointing of a task using barriers, we need to know all barriers it is waiting for. This information needs to be stored along with the task provate state checkpoint. The DDV for tasks needs to be augmented to track interactions with barriers.

5 Conclusions

In this paper we presented the issues posed by two common synchronization primitives in a sequential consistency DSM system, in backward error recovery from failures. We have tried to extend our earlier work[1] on dependency tracking and recovery line computation for DSM systems. We notice that there are *two* issues in the recovery - causality related dependencies and resource contention. We have noticed that while locks do not introduce any causality dependencies, barriers do, and information can be incorporated in the traditional checkpoint graph to aid causality related recovery line computation. Both locks and barriers introduce resource contention conflicts and these need to be resolved by heuristics. Further we note that because of these resource related conflicts a unique latest recovery line is not defined. This necessitates the use of heuristics.

In the process of the above description we have described what support in the form of mechanisms from IPC implementation we need in order to support checkpointing and

recovery. Also the additional data structures required for checkpointing and recovery have been described.

One may also conclude from our description that uncoordinated checkpointing poses additional challenges (due to resource conflicts) when dealing with synchronization primitives in shared memory applications. This makes the case for coordinated checkpointing in clusters stronger.

An obvious direction for research is to see what constraints are imposed by the other synchronization primitives used in shared memory applications. The ideas discussed here will be applied to our implementation of checkpointing and restart of applications in Kerrighed.

References

[1] R. Badrinath and C. Morin. Common Mechanisms For Supporting Fault Tolerance in DSM and Message Passing Systems. Technical Report RR-4613, INRIA, November 2002. http://www.inria.fr/rrrt/rr-4613.html.

[2] R. Badrinath, C. Morin, and G. Vallée. Checkpointing and Recovery of Shared Memory Parallel Applications in a Cluster. In *Proceedings of the DSM03 Workshop/ IEEE CCGRID 2003 Conference* (to appear), May 2003.

[3] R. Baldoni, G. Cioffi, J. Helary, and M. Raynal. Direct dependency-based determination of consistent global checkpoints. *International Journal of Computer Systems Sciences and Engineering*, 16(1):43–49, 2001.

[4] G. Cabillic, G. Muller, and I. Puaut. The performance of consistent checkpointing in distributed shared memory systems. In *Symposium on Reliable Distributed Systems*, pages 96–105, September 1995.

[5] R. Christodoulopoulou and A. Bilas. Dynamic data replication for tolerating single node failures in shared virtual memory clusters of workstations, June 2001.

[6] M. Costa, P. Guedes, M.Sequeira, N. Neves, and M. Castro. Lightweight logging for lazy release consistent distributed shared memory. In *Operating Systems Design and Implementation*, pages 59–73, 1996.

[7] M. Elnozahy, L. Alvisi, Y-M. Wang, and D.B. Johnson. A survey of rollback-recovery protocols in message-passing systems. Technical Report CMU-CS-99-148, Carnegie Mellon University, June 1999. Appears also in Computing Surveys, September, 2002.

[8] G. Janakiraman and Y. Tamir. Coordinated checkpointing-rollback error recovery for distributed shared memory multicomputers. In *Symposium on Reliable Distributed Systems*, pages 42–51, 1994.

[9] B. Janssens and W.K. Fuchs. Reducing interprocessor dependence in recoverable distributed shared memory. In *Symposium on Reliable Distributed Systems*, pages 34–41, 1994.

[10] A.-M. Kermarrec, C. Morin, and M. Banâtre. Design, implementation and evaluation of ICARE: an efficicent recoverable DSM. *Software Practice and Experience*, 28(9):981–1010, July 1998.

[11] A. Kongmunvattana, S. Tanchatchawal, and Nian-Feng Tzeng. Coherence-based coordinated checkpointing for software distributed shared memory systems. In *International Conference on Distributed Computing Systems*, pages 556–563, 2000.

[12] C. Morin and I. Puaut. A survey of recoverable distributed shared virtual memory systems. *IEEE Transaction on Parallel and Distributed Systems*, 8(9):959–969, 1997.

[13] F. Sultan, T. Nguyan, and L. Iftode. Scalable fault tolerant distributed shared memory. In *Supercomputing 2000 High Performance Networking and Computing Conference*, Nov. 2000.

[14] G. Suri, R. Janssens, and W. K. Fuchs. Reduced overhead logging for rollback recovery in distributed shared memory. In *International Symposium on Fault-Tolerant Computing Systems*, pages 279–288, Pasadena, California, 1995.

[15] Y.M. Wang, A. Lowry, and W.K. Fuchs. Consistent global checkpoints based on direct dependency tracking. *Information Processing Letters*, 50:223–230, 1994.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399